Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 52 (2015) 530 – 537

Procedia Computer Science

6th International Conference on Ambient Systems, Networks and Technologies (ANT 2015)

Using Category Theory to Verify Implementation Against Design in Concurrent Systems

Ming Zhu[a], Peter Grogono[a], Olga Ormandjieva[a]

[a]*Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada*

Abstract

The research has shown that process-oriented programming languages provide a suitable means for developing concurrent systems. However, in the development of a concurrent system, there is a challenge to manage consistency between design and implementation. To deal with such a challenge, we propose a new formal verification methodology and illustrate it by a running example. In this methodology, a concurrent system is designed using a process algebra, namely communicating sequential processes, and implemented in a process-oriented programming language, namely Erasmus. The consistency between the design and the implementation of such a concurrent system is verified formally using category theory.

Peer-review under responsibility of the Conference Program Chairs

Keywords: category theory; concurrent system; CSP; Erasmus; process-oriented programming; verification

1. Introduction

A concurrent system usually involves several interactive processes communicating simultaneously, which may lead to the exhibition of a large number of different behaviors in such a system. Incorporating knowledge and experience to manage design and implementation of concurrent systems is considered a serious challenge [1]. Adding formality to the system development process has a great benefit of reducing, or even preventing, the introduction of errors in the design and the implementation of a concurrent system.[2]. However, formal verification of a concurrent program against its design imposes its own challenges, including the cost for learning various mathematical notations and techniques. The goal of this paper is to provide a categorical basis for verifying implementation against design in a concurrent system developed in *Erasmus*, a process-oriented programming language. The objectives of the research are threefold: (1) to formalize the design of a concurrent system using *Communicating Sequential Processes*(CSP)[3,4], and analyze traces of events from the design, (2) to implement the concurrent system using Erasmus, and analyze

* Ming Zhu. Tel.: +1-514-848-2424 Ext.7149 ; fax: +1-514-848-2830.
E-mail address: zhu_ming@encs.concordia.ca

Peer-review under responsibility of the Conference Program Chairs
doi:10.1016/j.procs.2015.05.030

traces of events from abstraction of implementation, and (3) to verify implementation against design using *category theory*.

The rest of the paper is organized as follows. Section 2 provides some background on the process-oriented programming language Erasmus and the category theory, and discusses the related work. In Section 3 introduces the methodology for verifying the implementation in Erasmus against the design in CSP. The methodology is illustrated on a running example. Section 4 concludes the paper and suggests directions for future research work.

2. Background and Related Work

2.1. Process-Oriented Programming Languages

Process-oriented programming languages are likely to be the next programming paradigm[5]. Many process-oriented programming languages are based on process algebra CSP and π-Calculus. Process-oriented programming is based on processes that communicate by passing messages through channels rather than objects invoking one another's methods in object-oriented programming[1,6]. Process-oriented programming satisfies several requirements: safe concurrency, scalability, evolvability, and weak coupling between components[1].

To support process-oriented programming, several languages and libraries are designed, like *Erasmus*[7],*occam-π*[8] and JCSP for Java[9]. Though JCSP provides processes and channels for computation and passing messages, it is still a library added to object-oriented programming language Java. *occam-π* programs are constructed as process networks with processes as nodes and channels as edges for passing message. A channel is typed to specify the kinds of messages that can be passed through itself, while a protocol is defined to specify a sequence of messages that can be passed through a channel. Besides, in *occam-π*, a lower-level process network can be abstracted as a node in a higher-level process network, which conforms to the software engineering principle: separation of concerns. Compared with *occam-π*, Erasmus has similar features, but it provides more by including a notion port. A port that is of the type of a protocol works as an interface of a process to connect to a channel. A process can have several ports. Each port of a process specifies the types and the sequences of messages the process and receive or send through a channel. With the notion of port, it helps to specify and analyze passing messages between processes and channels. In this research, Erasmus is chosen to implement concurrent systems.

2.2. Category Theory

As category theory is helpful towards discovering and verifying connections in different areas with preserving structures in those areas[10,11], it has been proposed as a conceptual framework to formalize refinement from design to implementation across languages of various kinds [12,13]. However, for the analysis of implementation of concurrent systems by process-oriented languages, not much research has been done yet with category theory. To explore this research area, a verification approach based on category theory and data flow analysis is proposed to check whether some properties of concurrent systems in the design are preserved in the implementation[14]. As an extension and a continual work, this paper keeps working on verification between design and implementation of concurrent systems by process-oriented languages.

To understand the content of this paper, some of the categorical constructs are listed below:

- A *category* consists of *objects* and *morphisms*. A morphism $f : A \to B$ has object A as its *domain* and object B as its *codomain*, respectively. If there are morphisms $f : A \to B$ and $g : B \to C$, then there is also a morphism $g \circ f : A \to C$ called their *composition*. Composition is *associative*: $(h \circ g) \circ f = h \circ (g \circ f)$. Every object X has an *identity morphism* Id_X. For every morphism $f : A \to B$, $Id_B \circ f = f = f \circ Id_A$.
- A *functor* $F : C \to D$ maps each object of category C onto a corresponding object of category D, and maps each morphism of category C onto a corresponding morphism of category D, with preserving structure and composition.

3. Methodology

The proposed methodology for verifying implementation against design in concurrent systems consists of the following steps:

1. **Designing**: (a) Model the conceptual design of each process and the concurrent system by CSP. (b) Generate and analyze the traces of events from each process and the concurrent system.
2. **Implementing**: (a) Implement each process and the concurrent system by Erasmus. (b) Abstract implementation to focus on events from processes and the concurrent system. (c) Generate and analyze the traces of events based on the abstraction of implementation.
3. **Verifying**: (a) Construct categories based on the traces of events from design. (b) Construct categories based on the traces of events from implementation. (c) Construct functors to verify implementation against design.

This methodology is illustrated in a running example.

3.1. An Overview of the Example

In this example, there are three processes *Student*, *TeachingAssistant* and *Professor*. They collaborate as a concurrent system to deal with questions and answers as the following steps:

1. *Student* asks *TeachingAssistant* a question. *Student* has to wait for the answer before asking another question.
2. If *TeachingAssistant* can answer the question, the answer will be given to *Student*. Otherwise, *TeachingAssistant* will forward the question to *Professor*.
3. Once *Professor* received the question, it will give the answer to *TeachingAssistant*, and then *TeachingAssistant* will forward the answer to *Student*.
4. steps 1,2,3 can repeat indefinitely.

In the requirements, there are two scenarios. One is that the *TeachingAssistant* can answer the question, the other is that *Professor* helps *TeachingAssistant* to answer the question.

3.2. Step 1: Designing

The aim of this step is to design and analyze the processes and the concurrent system by CSP based on the textual description of the system requirements.

3.2.1. Step1.a: Model the Conceptual Design

As CSP can model and specify processes in concurrent system, for this example, the design of the above described system is specified as follows:

$$Stud = s.q \rightarrow t.a \rightarrow Stud, \qquad Prof = t.q \rightarrow p.a \rightarrow Prof,$$
$$TA = ((s.q \rightarrow t.a \rightarrow TA) \sqcap (s.q \rightarrow t.q \rightarrow TA)) \,\Box\, (p.a \rightarrow t.a \rightarrow TA).$$

In this design, event $s.q$ indicates the question asked by *Student* to *TeachingAssistant*; event $t.a$ represents the answer given by *TeachingAssistant* to *Student*; event $t.q$ stands for the question forwarded by *TeachingAssistant* to *Professor*; event $p.a$ describes the answer given by *Professor* to *TeachingAssistant*; $\rightarrow$ denotes the "occurs before" relation between events; $\sqcap$ means the nondeterministic choices made by the process itself; and $\Box$ stands for the deterministic choices based on the event from the environment.

3.2.2. Step1.b: Generate and Analyze the Traces

To understand the behaviors of a concurrent system, traces are used for analysis in CSP. A trace of events represents a sequential record of the behavior of a process. A process behaves in different ways leading to different traces of events.

For the abovementioned example, all possible traces of each process *Student*,*TeachingAssistant*, and *Professor* can be generated, analyzed and represented from the CSP specification of the design as follows:

$$traces(Stud) = \{\langle\rangle, \langle s.q\rangle\} \cup \{\langle s.q, t.a\rangle^\frown t \mid t \in traces(Stud)\}, \quad traces(Prof) = \{\langle\rangle, \langle t.q\rangle\} \cup \{\langle t.q, p.a\rangle^\frown t \mid t \in traces(Prof)\},$$

$$traces(TA) = \{\{\langle\rangle, \langle s.q\rangle\} \cup \{\langle s.q, t.a\rangle^\frown t \mid t \in traces(TA)\}\} \cup \{\{\langle\rangle, \langle s.q\rangle\} \cup \{\langle s.q, t.q\rangle^\frown t \mid t \in traces(TA)\}\}$$
$$\cup \{\{\langle\rangle, \langle p.a\rangle\} \cup \{\langle p.a, t.a\rangle^\frown t \mid t \in traces(TA)\}\}.$$

In this listing of traces, the function *traces*() stands for generating a set of all possible traces; t in $t \in traces(P)$ is one of the traces of process P; $\langle event_1, \cdots, event_n\rangle$ indicates the a specific trace of events; $^\frown$ concatenates two traces into one; and $\{traces_1\} \cup \{traces_2\}$ denotes the process may behave as either $\{traces_1\}$ or $\{traces_2\}$.

When processes *Student*, *TeachingAssistant*, and *Professor* work in parallel as a system, CSP operator "||" models communication between processes. According to CSP, if there is a communication between two processes, there must be an event that occurs in both processes simultaneously. The set of all possible traces of the system can be generated, analyzed and represented from the CSP specification of the design as follows:

$$traces(Stud \parallel TA \parallel Prof) = \{\langle\rangle, \langle s.q\rangle\} \cup \{\langle s.q, t.a\rangle^\frown t \mid t \in traces(Stud \parallel TA \parallel Prof)\}$$
$$\cup \{\langle\rangle, \langle s.q\rangle, \langle s.q, t.q\rangle, \langle s.q, t.q, p.a\rangle\} \cup \{\langle s.q, t.q, p.a, t.a\rangle^\frown t \mid t \in traces(Stud \parallel TA \parallel Prof)\}$$

According to the generated traces of events of processes running in parallel, the system should behave as either *TeachingAssistant* answers the question from *Student* directly, or *TeachingAssistant* asks help from *Professor* to answer *Student*.

3.3. Step2: Implementing

The aim of this step is to implement the processes and the concurrent system by Erasmus based on the design, and analyze the behaviors of the systems based on traces of events generated from abstraction of implementation.

```
Prot = protocol { question | answer }    //accept question or answer

Student= process -s:Prot, +t:Prot {
loop {
  s.question;      //ask the question via port s
  t.answer;        //receive the answer via port t
 } }

TeachingAssistant = process +s:Prot, -t:Prot, +p:Prot, -t':Prot {
loop
 select{             //deterministic choices depend on the environment
  ||s.question;      //receive the question from Student via port s
    case{            //nondeterministic choices made by the process
     |canAnswer| then t.answer;      //send the answer to Student via port t
     || t'.question; }               //ask the question to Professor via port t'
  ||p.answer;       //receive the answer from Professor via port p
    t.answer;       //send the answer to Student via port t
 } }

Professor = process +t':Prot, -p:Prot {
loop{
  t'.question;      //receive the question from TeachingAssistant via port t'
  p.answer;         //send the answer to TeachingAssistant via port p
 } }
```

```
System = cell{       //encapsulate processes
  SQuestion, TAnswer, T'Question, PAnswer: Prot;       // channels to connect ports
  Student(SQuestion,TAnswer);
  TeachingAssistant(SQuestion,TAnswer,PAnswer,T'Question);
  Professor(T'Question,PAnswer);
}
```

In this implementation, there are two scenarios: *TeachingAssistant* answering the question from *Student*, and *TeachingAssistant* resorting to help from *Professor* to answer the question from *Student*. To communicate with each other, two processes need to build a channel between their ports. For example, process *Student* can ask a *question* through port *s*, then the *question* passes through the channel *SQuestion*, and the *question* is received on port *s* by process *TeachingAssistant*.

3.3.1. Step2.b: Abstract the Implementation

Since the interest in this paper is in analyzing the behaviors of the system based on traces of events, an abstraction is created for extracting the code pertaining to generate traces of events. The abstraction of implementation contains loops, deterministic choices, nondeterministic choices, sending and receiving messages through ports.

The abstraction of the Erasmus implementation is provided as follows:

$$\begin{aligned} Stud &= \textbf{loop}\ \{Stud.s.q; Stud.t.a\},\\ TA &= \textbf{loop select}\ \{(TA.s.q; \textbf{case}\ \{TA.t.a \mid TA.t'.q\}) \mid (TA.p.a; TA.t.a)\},\\ Prof &= \textbf{loop}\ \{Prof.t'.q; Prof.p.a\}. \end{aligned}$$

where **loop** represents recursion; **select** together with | represent deterministic choices; **case** together with | represent nondeterministic choices; the notation *PROCESS.port.message*(for example *TA.s.q*) represents *message*(*question*) that occurs in *PROCESS*(*TA*) through *port*(*s*); and the symbol ";" is the delimiter to indicate the "occurs before" relation between messages.

3.3.2. Step2.c: Generate and Analyze the Traces

Although the syntax of Erasmus is different from CSP, the semantics of Erasmus is analogous to CSP. Some notations that model traces of events in CSP can be also used to model traces of events in Erasmus with preserving the same syntax and semantics, which includes $\frown$, $\cup$, $\langle\rangle$, $\sqcap$ and $\Box$. Like CSP, *traces* in Erasmus does not distinguish $\sqcap$ from $\Box$ in terms of traces of events occurred.

To generate and analyze the traces of processes, the function *traces*() in Erasmus are defined as follows:

$$\begin{aligned} &traces(P.pt.m) = \{\langle\rangle, \langle P.pt.m\rangle\},\\ &traces(P.pt.m_1; P.pt.m_2) = \{\langle\rangle, \langle P.pt.m_1\rangle, \langle P.pt.m_1, P.pt.m_2\rangle\},\\ &traces(\textbf{loop}\{P.pt.m\}) = \{\langle\rangle\} \cup \{\langle P.pt.m\rangle \frown t \mid t \in traces(\textbf{loop}\{P.pt.m\})\},\\ &traces(\textbf{case}\ \{P.pt.m_1 \mid \cdots \mid P.pt.m_n\}) = traces(P.pt.m_1) \cup \cdots \cup traces(P.pt.m_n),\\ &traces(\textbf{select}\ \{P.pt.m_1 \mid \cdots \mid P.pt.m_n\}) = traces(P.pt.m_1) \cup \cdots \cup traces(P.pt.m_n). \end{aligned}$$

For each process in the abstract implementation, the traces of events are generated and analyzed as follows:

$$\begin{aligned} traces(Stud) &= traces(\textbf{loop}\ \{Stud.s.q; Stud.t.a\})\\ &= \{\langle\rangle, \langle Stud.s.q\rangle\} \cup \{\langle Stud.s.q, Stud.t.a\rangle \frown t \mid t \in traces(Stud)\},\\ traces(TA) &= traces(\textbf{loop select}\ \{(TA.s.q; \textbf{case}\ \{TA.t.a \mid TA.t'.q\}) \mid (TA.p.a; TA.t.a)\})\\ &= traces(\textbf{loop}\{(TA.s.q; (TA.t.a \sqcap TA.t'.q))\Box(TA.p.a; TA.t.a)\})\\ &= \{\{\langle\rangle, \langle TA.s.q\rangle\} \cup \{\langle TA.s.q, TA.t.a\rangle \frown t \mid t \in traces(TA)\}\}\\ &\quad \cup \{\{\langle\rangle, \langle TA.s.q\rangle\} \cup \{\langle TA.s.q, TA.t'.q\rangle \frown t \mid t \in traces(TA)\}\} \end{aligned}$$

$$\cup \{\{\langle\rangle, \langle TA.p.a\rangle\} \cup \{\langle TA.p.a, TA.t.a\rangle^\frown t \mid t \in traces(TA)\}\},$$

$$traces(Prof) = traces(\textbf{loop}\ \{Prof.t'.q; Prof.p.a\})$$

$$=\{\langle\rangle, \langle Prof.t'.q\rangle\} \cup \{\langle Prof.t'.q, Prof.p.a\rangle^\frown t \mid t \in traces(Prof)\}.$$

In the implementation, when one process communicates with another process, an event occurs as a pair such as ($PROCESS_1.port_1.message$, $PROCESS_2.port_2.message$) during one communication. This pair indicates $PROCESS_1$ sends a *message* through $port_1$, and $PROCESS_2$ receives the *message* through $port_2$. In the above implementation of the example, ports with the same name in different processes are connected by a channel. For example, (*Stud.s.q*,*TA.s.q*) is an event, which indicates *Stud* sends a *question* through its port *s*, *TA* receives the *question* through its port *s*, and ports are connected by the channel *SQuestion* according to the implementation.

To generate and analyze the traces of concurrent systems in the implementation, the function *traces*() together with the symbol || are defined as follows:

1. Given a process P with port pt_1 and a process Q with port pt_2, pt_1 and pt_2 are connected to a channel *ch*. P has a trace p, and Q has a trace q. The head of p and q are events p_0 and q_0 respectively, and the tail of p and q are traces p' and q' respectively. When p and q run in parallel,

$$traces(P \parallel Q) =\{\langle\rangle\} \cup \{\langle (P.pt_1.m_1, Q.pt_2.m_2)\rangle^\frown t \mid P.pt_1.m_1 = q_0, Q.pt_2.m_2 = p_0,$$
$$pt_1 \textit{ connected to } ch, pt_2 \textit{ connected to } ch, m_1 = m_2, t \in p' \parallel q'\}$$

2. Given processes $P_1, \cdots, P_n$, when they run in parallel as a system,

$$traces(P\|\cdots\|Q) = traces(P)\|\cdots\|traces(Q)$$

For each of the two scenarios for the system in the implementation, the traces of events are generated and analyzed as follows:

Scenario 1: *TeachingAssistant* answers the question

$$traces(Stud\|TA\|Prof) =\{\langle\rangle, \langle (Stud.s.q, TA.s.q)\rangle\}$$
$$\cup \{\langle (Stud.s.q, TA.s.q), (TA.t.a, Stud.t.a)\rangle^\frown s \mid s \in traces(Stud)\|traces(TA)\|traces(Prof)\}$$

Scenario 2: *Professor* helps *TeachingAssistant* to answer the question

$$traces(Stud\|TA\|Prof) =$$
$$\{\langle\rangle, \langle (Stud.s.q, TA.s.q)\rangle, \langle (Stud.s.q, TA.s.q), (TA.t'.q, Prof.t'.q)\rangle, \langle (Stud.s.q, TA.s.q), (TA.t'.q, Prof.t'.q), (Prof.p.a, TA.p.a)\rangle\}$$
$$\cup \{\langle (Stud.s.q, TA.s.q), (TA.t'.q, Prof.t'.q), (Prof.p.a, TA.p.a)(TA.t.a, Stud.t.a)\rangle^\frown s \mid s \in traces(Stud)\|traces(TA)\|traces(Prof)$$

3.4. Step3: Verifying

The aim of this step is to verify consistency between design and implementation by constructing categories and functors. In this research, consistency between the design and the implementation is defined as follows:

Given a sequence of traces of events in the design representing the progress of the system, $DSeq : \langle\rangle \to \langle devent_1\rangle \to \cdots \to \langle devent_1, \ldots, devent_n\rangle$, and a sequence of traces of events in the implementation representing the progress of the system, $ISeq : \langle\rangle \to \langle ievent_1\rangle \to \cdots \to \langle ievent_1, \ldots, ievent_n\rangle$. If there exist a mapping from *ISeq* to *DSeq* with structure preserved between traces of events, *ISeq* is consistent with *DSeq*. If all sequences in the design have corresponding mapping sequences in the implementation, the implementation of the system is consistent with the design of the system.

3.4.1. Step3.a: Construct the Category of Traces of Events from the Design

A category named **DEvents** captures the designed behaviors of the system based on traces of events extracted from the design in section 3.2. In **DEvents**, each object represents a trace of events of the system designed; each morphism models the prefix relationship between traces denoted by $\preccurlyeq$ to indicate the progress of the system; and each identity represents the prefix of a trace to itself.

Fig. 1, illustrates part of **DEvents** category with the first few traces of unbounded sequences.

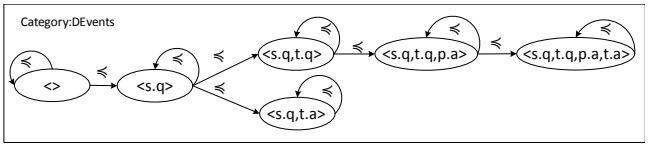

Fig. 1. Category of Traces from the Design

3.4.2. Step3.b: Construct the Category of Traces of Events from the Implementation

A category named **IEvents** captures the implemented behaviors of the system based on traces of events extracted from the abstraction in section 3.3. In **IEvents**, each object represents a trace of events of the system implemented; each morphism models the prefix relationship between traces denoted by $\preccurlyeq$ to indicate the progress of the system; and each identity represents the prefix of a trace to itself.

Fig. 2, illustrates part of **IEvents** category with the first few traces of unbounded sequences.

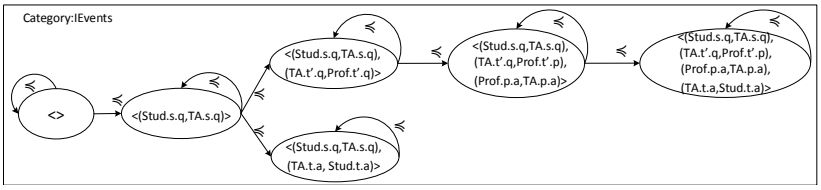

Fig. 2. Category of Traces from the Implementation

3.4.3. Step3.c: Construct the Functor for Verification

To verify the implementation against the design, the construction of a functor can be used. If there exists a functor that maps the category of the traces from implementation to the category of the traces from design, the implementation is consistent with the design. Otherwise, the implementation is inconsistent with the design.

Based on the analysis of categories **DEvents** and **IEvents**, the consistency between the design and the implementation is verified by constructing a functor **IToD**: **IEvents** $\rightarrow$ **DEvents**. This functor maps objects and morphisms of **IEvents** to the corresponding objects and morphisms of **DEvents** as follows:

- an object *oi* of **IEvents** maps to an object *od* of **DEvents**, when the trace in *oi* matches the trace in *od*. For example, $\langle(Stud.s.q, TA.s.q)\rangle$ in **IEvents** represents an event that *Student* sends a *question* to *TeachingAssistant*, and $\langle s.q\rangle$ in **DEvents** represents an event that *Student* sends a *question* to *TeachingAssistant*. Thus, $\langle(Stud.s.q, TA.s.q)\rangle$ matches $\langle s.question\rangle$.
- a morphism $mi : oi_1 \xrightarrow{\preccurlyeq} oi_2$ of **IEvents** maps to a morphism $md : od_1 \xrightarrow{\preccurlyeq} od_2$ of **DEvents**, when oi_1 and oi_2 match od_1 and od_2 respectively, and $\preccurlyeq$ from oi_1 to oi_2 matches $\preccurlyeq$ from od_1 to od_2. For example, $\langle(Stud.s.q, TA.s.q)\rangle \xrightarrow{\preccurlyeq} \langle(Stud.s.q, TA.s.q), (TA.t.a, Stud.t.a)\rangle$ maps to $\langle s.q\rangle \xrightarrow{\preccurlyeq} \langle s.q, t.a\rangle$.
- identities mapping and compositions of morphisms mapping are preserved.

Fig. 3, shows that **IToD**: **IEvents** $\rightarrow$ **DEvents** is a functor.

A successful construction of the functor **IToD** indicates that the implementation and the design are consistent.

4. Conclusion and Future Work

Verification based on category theory is a formal means for better understanding and analyzing implementation against design in concurrent systems developed by process-oriented languages such as Erasmus. In this paper, a methodology for categorically verifying consistency between design and implementation is presented.

This methodology is based on CSP, category theory and abstraction of implementation, and is illustrated by a running system with processes *Student*, *TeachingAssistant* and *Professor* executing in parallel. In doing so, the design

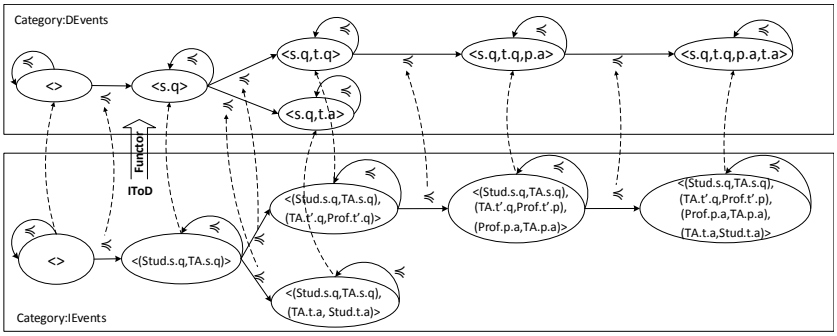

Fig. 3. Functor IToD

of the system is modeled and analyzed by CSP, the implementation of the system is created by Erasmus, traces of events of the implementation are analyzed based on abstraction, categories of traces of events from the design and implementation are created, and, by constructing a functor, the consistency between the design and the implementation is verified.

The work presented in this paper is preliminary, but provides a number of avenues of research interest. For example, it would be of interest to analyze more complex examples that can scale up to realistic concurrent systems. It would also be useful to explore the application of other categorical structures for verification.

Acknowledgements

We would like to gratefully acknowledge Dr. Pankaj Kampthan for his helpful comments and suggestions on the manuscript.

References

1. P. Grogono and B. Shearing. Concurrent software engineering: Preparing for paradigm shift. In *Proceedings of the First C* Conference on Computer Science and Software Engineering*, pages 99–108, Montreal, Canada, 2008.
2. P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, Secaucus, United States, 1996.
3. C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Englewood Cliffs, United States, 1985.
4. A.W. Roscoe. *Understanding concurrent systems*. Springer, London, United Kingdom, 2010.
5. P. Welch. Life of occam-pi. In *Communicating Process Architectures 2013*. Open Channel Publishing Ltd., 2013.
6. A. T. Sampson. *Process-oriented patterns for concurrent software engineering*. PhD thesis, University of Kent, Kent, United Kingdom, 2008.
7. P. Grogono. The erasmus project: Process oriented programming. `http://users.encs.concordia.ca/~grogono/Erasmus/erasmus.html`.
8. Occam-π: blending the best of CSP and the pi-calculus. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.
9. P. Welch and N. Brown. Java communicating sequential processes. `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.
10. M. Barr and C. Wells. *Category theory for computing science*. Prentice-Hall, Upper Saddle River,United States, 2012.
11. J. A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.
12. J. L. Fiadeiro. *Categories for software engineering*. Springer Berlin Heidelberg, Germany, 2005.
13. C.A.R. Hoare. Notes on an approach to category theory for computer scientists. In *Proceedings of the NATO Advanced Study Institute on Constructive Methods in Computer Science*, volume 55. Springer Berlin Heidelberg, Germany, 1989.
14. M. Zhu, P. Grogono, O. Ormandjieva, and P. Kamthan. Using category theory and data flow analysis for modeling and verifying properties of communications in the process-oriented language erasmus. In *Proceedings of the Seventh C* Conference on Computer Science and Software Engineering*, pages 24:1–24:4, Montreal, Canada, 2014.